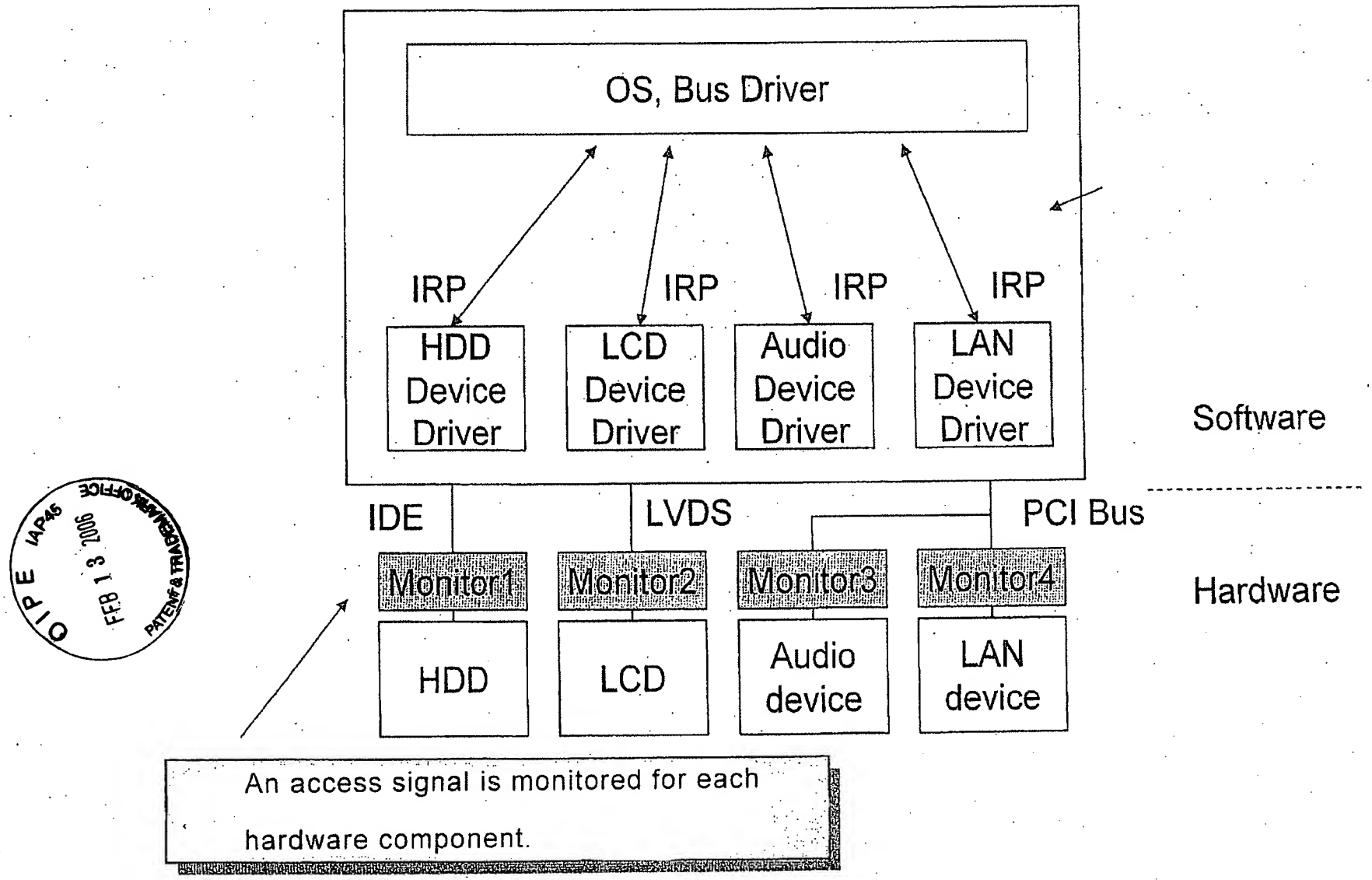
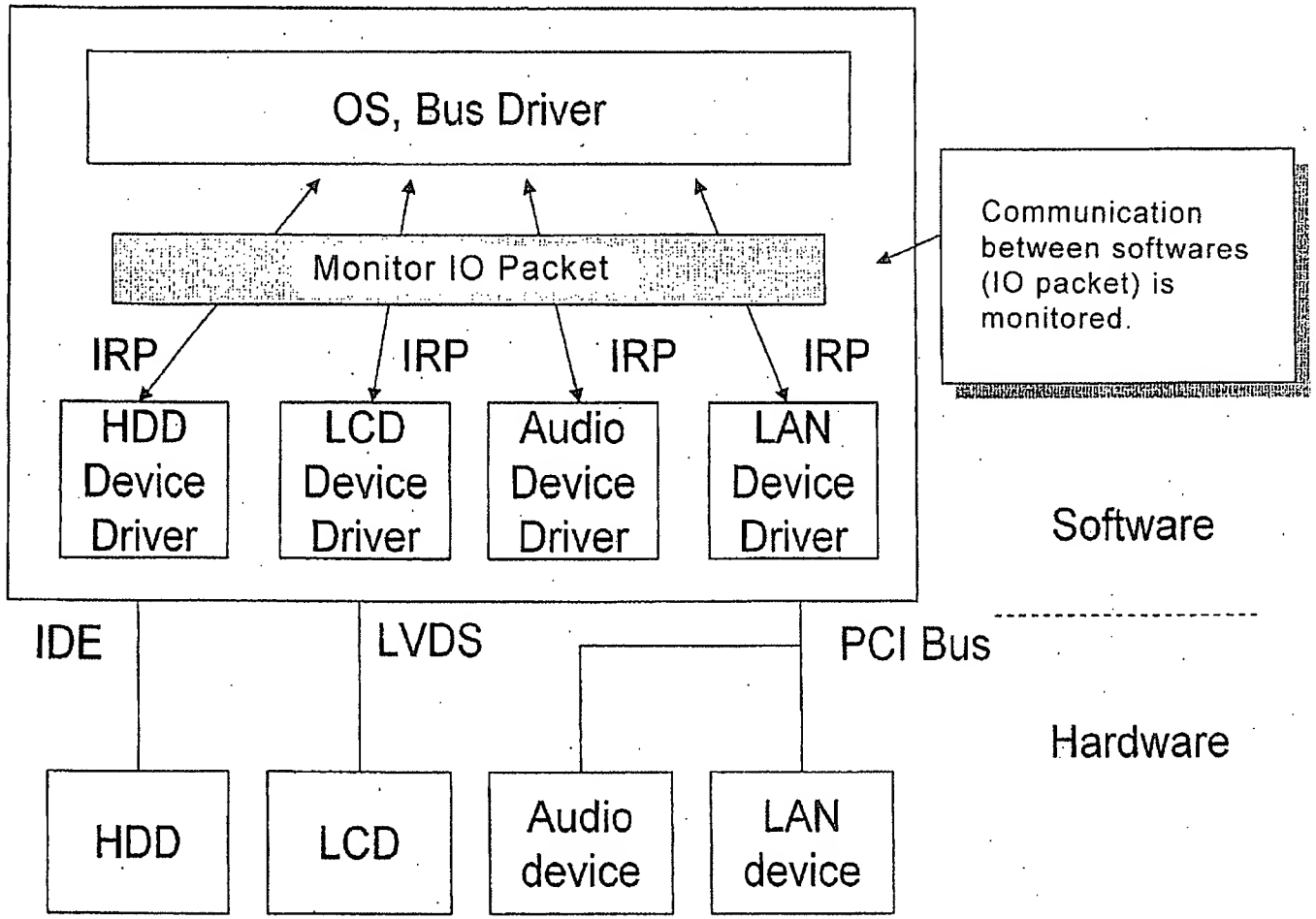
Hetzler

DIAGRAM A



Present Invention

DIAGRAM B



WinNTのI/Oパケット(1/4)

vate thread, which generates a context swap when control is transferred. For efficiency, most Kernel-mode support functions and drivers use Spin Locks for running at IRQL <= DISPATCH_LEVEL. We illustrate the use of Events extensively in Chapter 28 and Chapter 30.

Microsoft recommends having no more than one Spin Lock acquired at a time. This is sound advice if you wish to avoid *deadlock*, in which contention for two resources results in two different threads of execution waiting for each other.[4] This advice, however, is often unrealistic. This is because in order to get maximum throughput with your driver, you might need to have multiple Spin Locks that synchronize nominally independent resources. Only when these resources are shared is there a potential problem. Whenever multiple Spin Locks are needed, you should always *lock* resources in the *same* order and *unlock* them in the *inverse* order. To do otherwise is to invite deadlock, which is difficult to debug, both intellectually and from down in the bits using a debugger.

I/O Request Packets

The IRP (I/O request packet) is the basis of all transactions within the I/O system. It is the way the general I/O system talks to a top-level driver and how a top-level driver talks to a driver below itself. The last place an IRP visits is the actual driver level that talks to the physical device. One IRP at a high level may generate a sequence of IRPs at lower levels. For example, a request to open a file starts with a single IRP to the file system. However, before that IRP can actually open the file, it must read the root directory and each directory along the path until the desired directory is found. It might generate additional IRPs to "prime the pump" by prereading the initial part of the file, if the file system chooses to do this sort of optimization.

Without an IRP, there can be no I/O transaction. Therefore, no device can spontaneously interrupt a running application thread to notify the *application* that the device requires attention. Furthermore, an I/O request will always provide information precisely *once* to the application that called it. Only after you have managed to get an IRP into the system is there the possibility of interrupting the flow of a application thread by an asynchronous callback.

This suggests a discipline regarding how you write a driver: You don't actually enable interrupts on a device until you have an IRP, and you disable them after the IRP is completed. However, some devices may generate interrupts continuously for events that can be handled entirely within the driver. The *effect*, however, is that the device is doing no I/O unless an IRP is present.

IRPs are always allocated out of nonpaged memory. So if you have a pointer to an IRP, you can be sure that it is in memory and accessible. You cannot be as certain about the memory it *references*, but the IRP itself is always accessible.

Figure 5.1 shows a simplified I/O transaction. There, the IRP takes a simple—well, more-or-less simple—path. In reality, layered drivers can add substantial additional complexity. Some of the art of writing a driver is determining exactly what the layering looks like or where your driver fits into existing layers. In the figure and the discussion of it that follows, we've glossed over some critical details (which we look at in much greater depth throughout the book) so we don't confuse the issue with too much detail.

[4] For a more detailed discussion of deadlock, see Chapter 11, page 248; Chapter 14, page 294; and Chapter 17, page 350, which includes an example scenario.

(2/4)

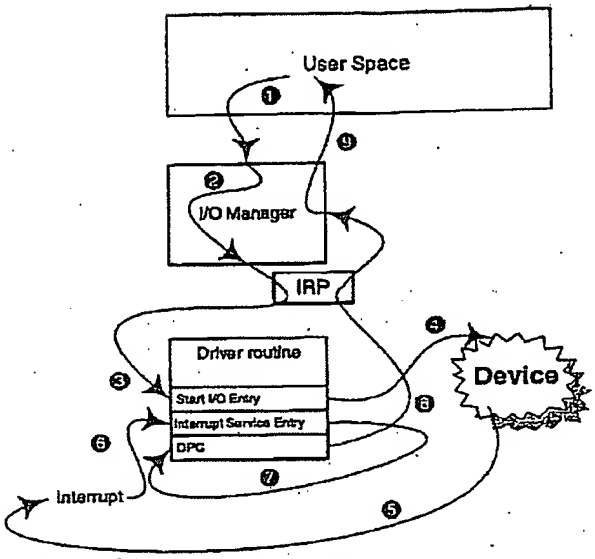


Figure 5.1: *Simplified IRP structure and I/O logic.*

- A call to the I/O system takes place in ❶. This call is routed to the I/O Manager.
- The I/O Manager allocates an IRP, initializes it, verifies the parameters (for example, it verifies that addresses are in the user's virtual address space), and then calls the driver's Dispatch Routine ❷.
- This routine calls the "Start I/O" entry point of the driver ❸.
- The driver initializes the I/O operation and activates the device ❹. At this point, if the I/O operation is a synchronous I/O operation, the user thread that initiated the I/O is suspended.
- Some time later, after the device completes its operation, the device uses an interrupt ❺ to report this situation.
- This interrupt is routed through the HAL and the kernel to the Interrupt Service entry point ❻.
- The Interrupt Service handler uses a DPC activated by the `DpcForIsr` function ❼ to set the necessary status code in the IRP and return control ❽ to the I/O Manager.
- The I/O Manager frees the IRP and uses the status to return the completion code to the user ❾.

We study all of these transactions in detail in Chapter 14.

Note that within the kernel space, I/O operations are *always* asynchronous, independent of the user-level availability of asynchronous I/O. The synchronicity perceived at the user level is an illusion. You will always program with the awareness of this asynchrony.

(3/4)

A single IRP can be passed from one driver layer to another during the I/O process. Rather than having to allocate a new IRP at every layer, you divide an IRP into two sections: the *fixed part* and a set of *I/O stack locations*. In the fixed part, or *header*, the I/O Manager works with information about the original I/O request as formulated by the user-to-kernel function call. This includes the caller's parameters, the address of the applicable Device Object, and some other state that we will discuss shortly. The fixed part also contains an *I/O status block*, which is used by drivers to report the status of the operation and, depending on the type of operation, the number of bytes transferred.

The highest-level driver has one or more *I/O stack locations* (there is always at least one). This location contains driver-specific parameters, such as a function code indicating the nature of the operation and other information. This is used by the highest-level driver to control its action. Each driver sets up the I/O stack location for the next lower-level driver, except, of course, for the lowest-level driver.

Each level of the driver can access its I/O stack location in the IRP. When an IRP is allocated for a monolithic driver, it has only one I/O stack location. When layered drivers are used (see Chapter 23), a driver can ask the driver it is going to call how many stack locations *it* requires. Then it can allocate an IRP with that value plus 1. The I/O Manager will do this for any IRPs it creates for you.

Once an IRP is allocated, the number of I/O stack locations cannot be changed. There is also no error checking done for bounds-checking, so if you allocate too few I/O stack locations, you will eventually crash the system.

Figure 5.2 illustrates how IRP packets use I/O stacks. It also illustrates some additional kernel subsystem interactions.

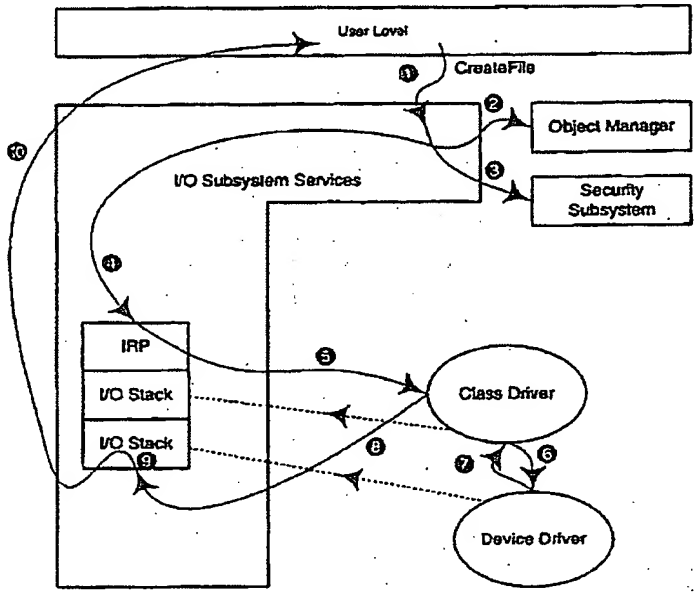


Figure 5.2: `CreateFile` *illustrating an IRP and use of I/O stacks.*

- The application performs a `CreateFile` operation ❶.
- The filename (in this case, the device name) is passed to the *Object Manager* ❷. This manager performs the necessary lookup in the Object Manager Namespace and determines which Device Object will be used for the device. If this fails, the `CreateFile` operation will fail.
- Once the device has been located, the I/O subsystem calls the *security subsystem* ❸ to verify that the calling thread has access to the device. Note that we did not say "that the user has access to the device" or "that the process has access to the device". These are orthogonal concepts in NT. A process can create a thread that has different security attributes than the process itself, and while the *process* might have access to the device, a thread might not necessarily have the same privileges.
- If the lookup and security check are successful, the I/O Manager prepares to call the driver and then allocates the IRP ❹. It knows, by using one of the fields in the Device Object structure, how many I/O stack locations are required.
- The I/O Manager then calls the Class Driver ❺, which performs any necessary work required to complete the `CreateFile` operation. This level uses the first I/O stack location, as shown.
- The Class Driver calls the Device Driver ❻, which performs *its* `CreateFile`-related work. Note that it uses, in addition to the fixed part of the IRP, the second I/O stack location. Before calling the Device Driver, however, the Class Driver has to set up the contents of this location for the lower-level driver.
- Finally, the Device Driver returns ❼ to the Class Driver, which may complete any initialization that may be based on the lower-level driver.
- The Class Driver then returns to the I/O Manager ❽, which deallocates the IRP ❾.
- The I/O Manager finally returns the status to the user ❿. The status is returned by the drive's setting the `Status` field (in the I/O stack location of the IRP). This will eventually cause the I/O Manager to call the kernel's internal equivalent of `SetLastError` before returning to the application. If the return code is other than `STATUS_SUCCESS`, the application gets a `FALSE` return value or some other similar failure indication; for example, `CreateFile` returns `INVALID_HANDLE_VALUE`. The application then calls `GetLastError` to determine the cause of the problem.[5]

Layered Drivers and IRPs

A higher-level driver can safely access only its own I/O stack location and the I/O stack location of its immediately lower driver. As the writer of a higher-level driver, you can make no assumptions about any layer other than the one immediately below your driver's layer. The number of layers can change dynamically (although infrequently); for example, when Filter Drivers are inserted or removed.

The lowest-level driver can safely access only its own I/O stack location and the common IRP fields.

Any driver that is added as a new layer can set its own completion function pointer into the IRP's I/O stack location. This function will be called by a lower-level driver and allows the layer

[5] The mapping between internal error codes and user-visible error codes is set out in Appendix B.